Lecture 05

For-Loops

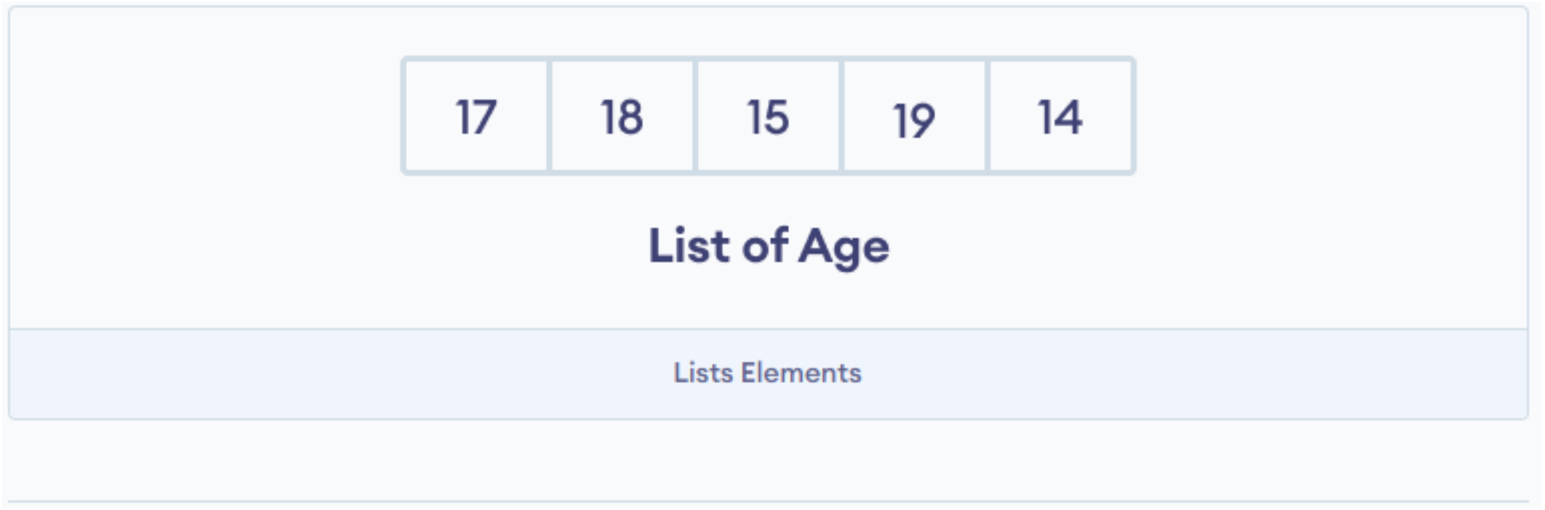
Annoucements

- Lab will be auto released on Wed
- Must know how to unzip/zip a folder (if you don’t know, make sure to study)
- Include today’s lecture
- Do not email any submission related to lab
- Absolutely no late submissions

What is a list in python?

In Python, lists are used to store multiple data at once.

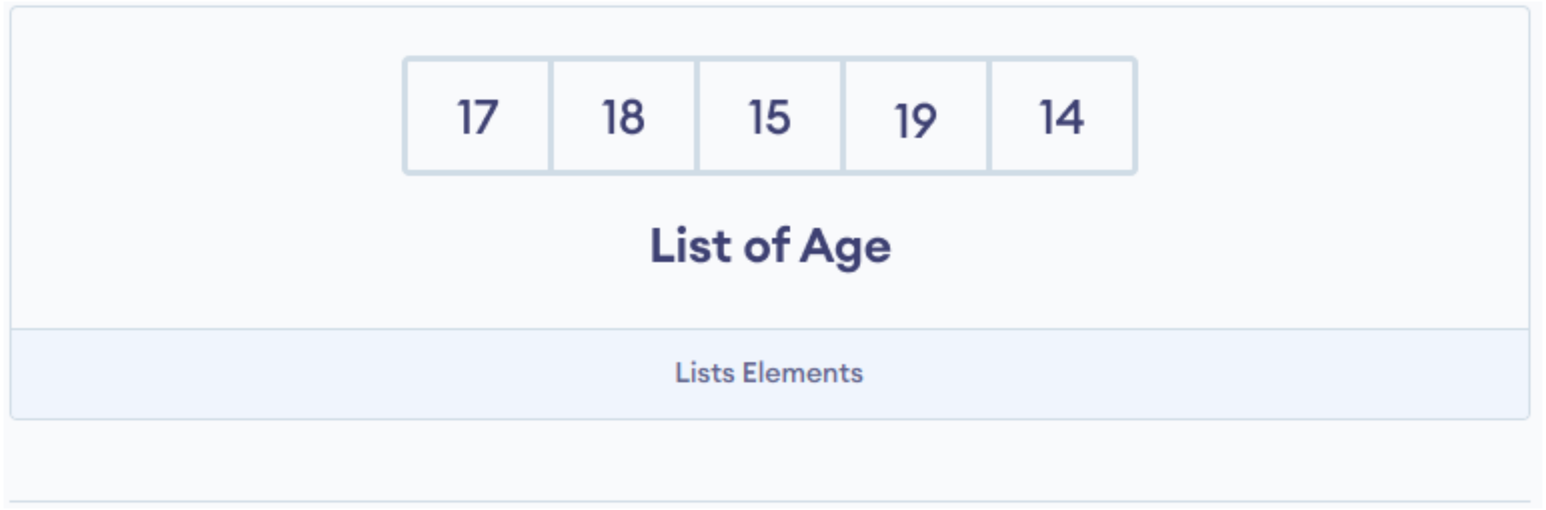
Suppose we need to record the ages of 5 students. Instead of creating 5 separate variables, we can simply create a list.



What is a list in python?

A list can store a collection of data of any size.

Python lists are one of the most versatile data types that allow us to work with multiple elements at once.



Creating a list

We create a list by placing elements inside [], separated by commas. For example,.

```
ages =  [19, 26, 23]
print(ages)
```

```
# Output: [19, 26, 23]
```

A list can

- Store elements of different types (integer, float, string, etc.)
- Store duplicate elements

```
# list with elements of different data types
list1 = [1, "Hello", 3.4]

# list with duplicate elements
list1 = [1, "Hello", 3.4, "Hello", 1]

# empty list
list3 = []
```

Accessing list elements

In Python, lists are ordered and each item in a list is associated with a number. The number is known as a list index.

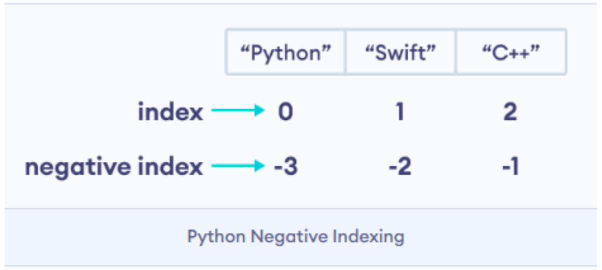
The index of the first element is 0, second element is 1 and so on. For example,

```
languages = ["Python", "Swift", "C++"]
# access item at index 0
print(languages[0])   # Python

# access item at index 2
print(languages[2])   # C++
```

Negative indexing in python

Python allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on.

```
languages = ["Python", "Swift", "C++"]
# access item at index 2
print(languages[-1])   # C++

# access item at index 0
print(languages[-3])   # Python
```



Add elements to a list

Lists are mutable (changeable). Meaning we can add and remove elements from a list.

Python list provides different methods to add items to a list.

The append() method adds an item at the end of the list. For example,

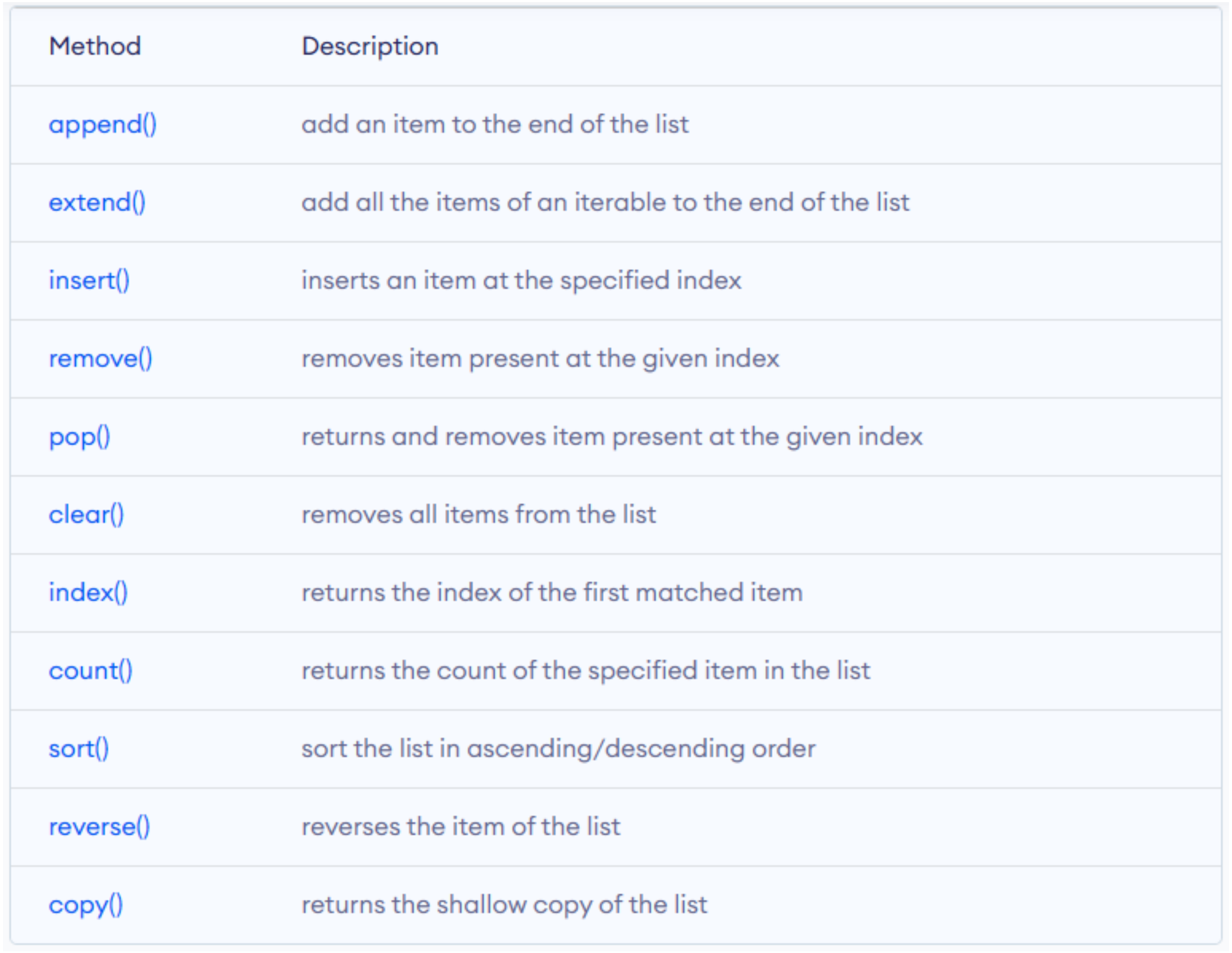
Add elements to a list

```
numbers = [21, 34, 54, 12]
print("Before Append:", numbers)

# using append method
numbers.append(32)

print("After Append:", numbers)
```

Output:

```
Before Append: [21, 34, 54, 12]
After Append: [21, 34, 54, 12, 32]
```

List methods



Method	Description
append()	add an item to the end of the list
extend()	add all the items of an iterable to the end of the list
insert()	inserts an item at the specified index
remove()	removes item present at the given index
pop()	returns and removes item present at the given index
clear()	removes all items from the list
index()	returns the index of the first matched item
count()	returns the count of the specified item in the list
sort()	sort the list in ascending/descending order
reverse()	reverses the item of the list
copy()	returns the shallow copy of the list

Example: Summing the Elements of a List

```
def sum(thelist):
    """Returns: the sum of all elements in thelist
    Precondition: thelist is a list of all numbers
    (either floats or ints)"""
    pass # Stub to be implemented
```

Remember our approach:
Outline first; then implement

Example: Summing the Elements of a List

```
def sum(thelist):
    """Returns: the sum of all elements in thelist
    Precondition: thelist is a list of all numbers
    (either floats or ints)"""
    # Create a variable to hold result (start at 0)
    # Add each list element to variable
    # Return the variable
```

Example: Summing the Elements of a List

```
def sum(thelist):
    """Returns: the sum of all elements in thelist
    Precondition: thelist is a list of all numbers
    (either floats or ints)"""
    result = 0
    result = result + thelist[0]
    result = result + thelist[1]
    …
    return result
```

There is a problem here

Working with Sequences

- Sequences are potentially **unbounded**
  - Number of elements inside them is not fixed
  - Functions must handle sequences of different lengths
  - **Example**: sum([1,2,3]) vs. sum([4,5,6,7,8,9,10])
- Cannot process with **fixed** number of lines
  - Each line of code can handle at most one element
  - What if # of elements > # of lines of code?
- We need a new **control structure**

The For-Loop

```
# Create local var x
x = seqn[0]
print(x)
x = seqn[1]
print(x)
...
x = seqn[len(seqn)-1]
print(x)
```

Not valid Python

```
# Write as a for-loop
for x in seqn:
    print(x)
```

Key Concepts

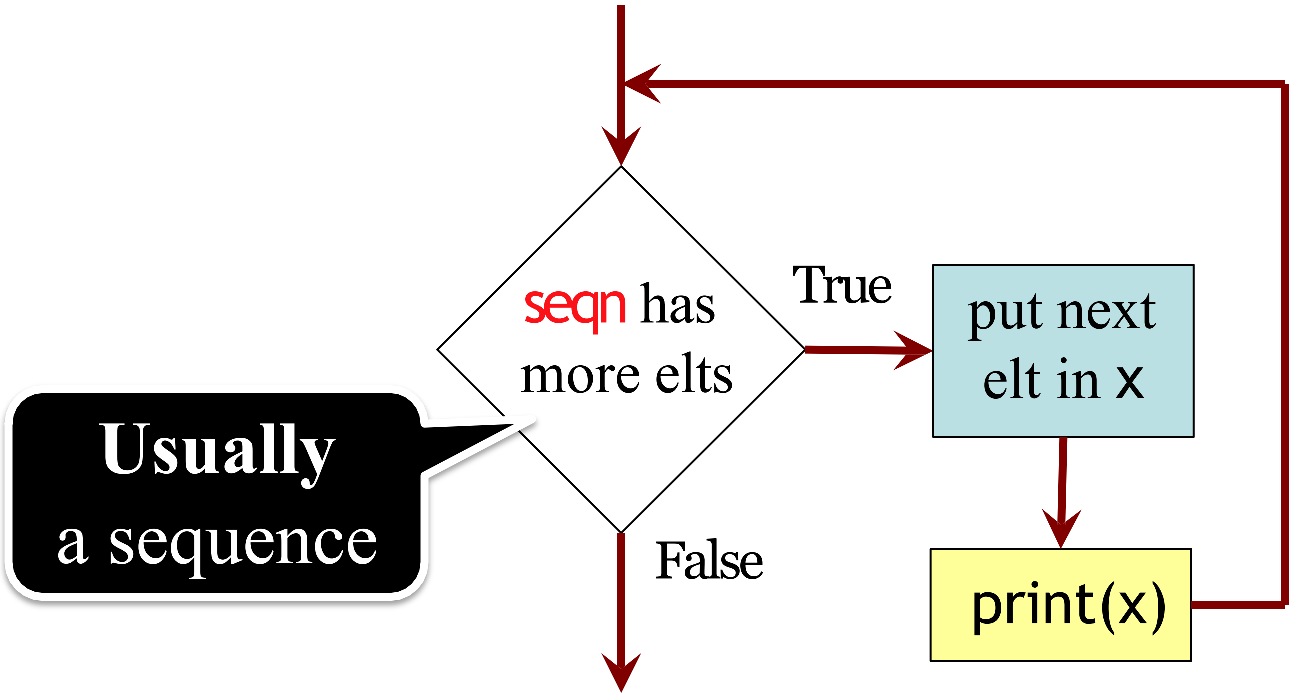
- **iterable**: seqn
- **loop variable**: x
- **body**: print(x)

Executing a For-Loop

The for-loop:

```
for x in seqn:
    print(x)
```

- **iterable:** seqn
- **loop variable:** x
- **body:** print(x)



Example: Summing the Elements of a List

```
def sum(thelist):
    """Returns: the sum of all elements in thelist
    Precondition: thelist is a list of all numbers
    (either floats or ints)"""
    # Create a variable to hold result (start at 0)
    # Add each list element to variable
    # Return the variable
```

Example: Summing the Elements of a List

```
def sum(thelist):
    """Returns: the sum of all elements in thelist
    Precondition: thelist is a list of all numbers
    (either floats or ints)"""
    result = 0

    for x in thelist:
        result = result + x

    return result
```

- **iterable:** thelist
- **loop variable**: x
- **body**: result=result+x

Example: Summing the Elements of a List

```
def sum(thelist):
    """Returns: the sum of all elements in thelist
    Precondition: thelist is a list of all numbers
    (either floats or ints)"""
    result = 0

    for x in thelist:
        result = result + x

    return result
```

Accumulator variable

- **iterable: thelist**
- **loop variable**: x
- **body**: result=result+x

The Accumulator

- In a slides saw the **accumulator**
  - Variable to hold a final (numeric) answer
  - For-loop added to variable at each step
- This is a common *design pattern*
  - Popular way to compute statistics
  - Counting, averaging, etc.
- It is not just limited to numbers
  - Works on **every type that can be added**
  - This means **strings**, **lists** and **tuples**!

Example: String-Based Accumulator

```
def despace(s):
    """Returns: s but with its spaces removed
    Precondition: s is a string"""
    # Create an empty string accumulator
    # For each character x of s
        # Check if x is a space
        # Add it to accumulator if not
```

Example: String-Based Accumulator

```
def despace(s):
    """Returns: s but with its spaces removed
    Precondition: s is a string"""
    result = ''
    for x in s:
        if x != ' ':
            result = result+x
    return result
```

Body

Modifying the Contents of a List

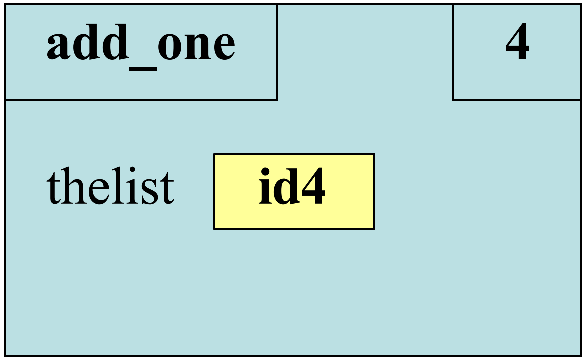
```
def add_one(thelist):
    """(Procedure) Adds 1 to every element in the list
    Precondition: thelist is a list of all numbers
    (either floats or ints)"""
    for x in thelist:
        x = x+1
    # procedure; no return
```

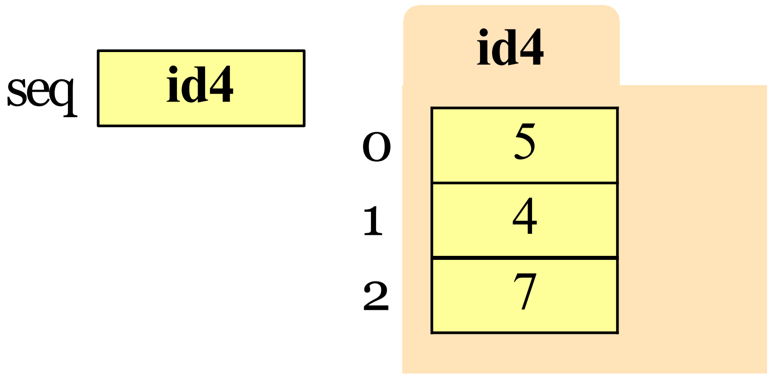
DOES NOT WORK!

For Loops and Call Frames

```
1.  def add_one(thelist):
2.      """Adds 1 to every elt
3.      Pre: thelist all nums"""
4.      for x in thelist:
5.          x = x+1
```

add_one(seq):

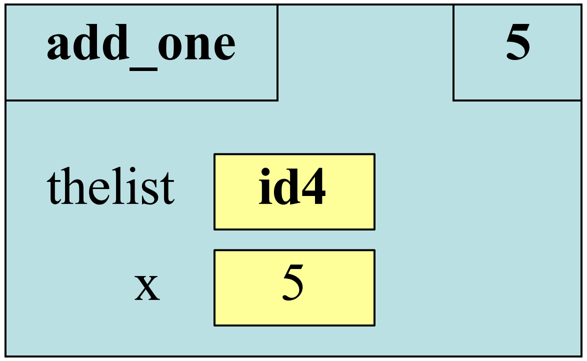


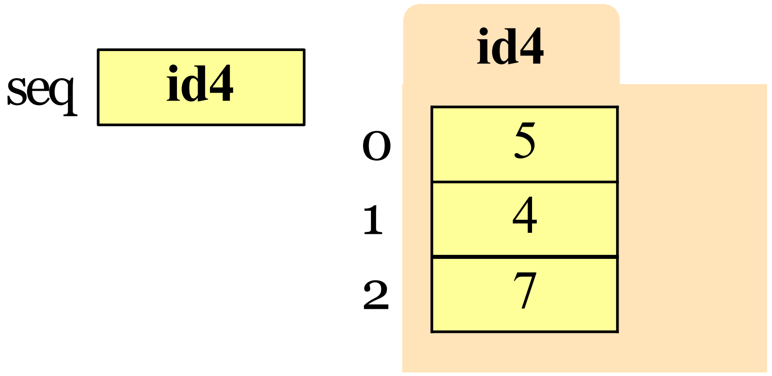


For Loops and Call Frames

```
1.  def add_one(thelist):
2.      """Adds 1 to every elt
3.      Pre: thelist all nums"""
4.      for x in thelist:
5.          x = x+1
```

add_one(seq):

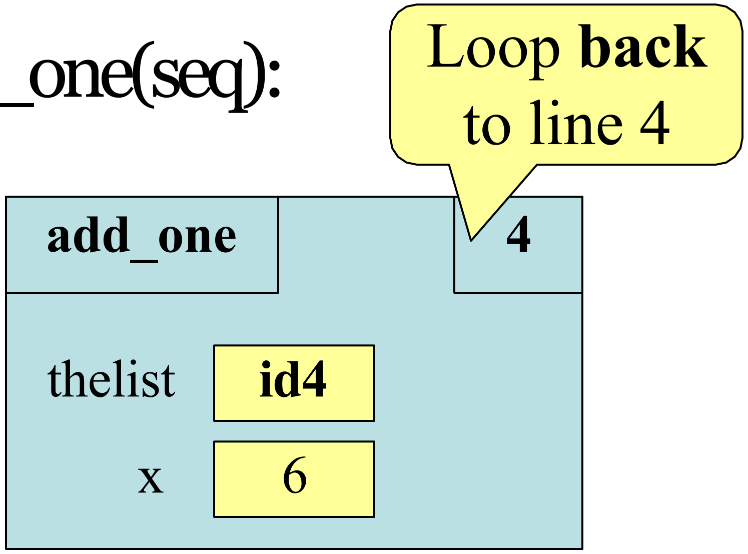


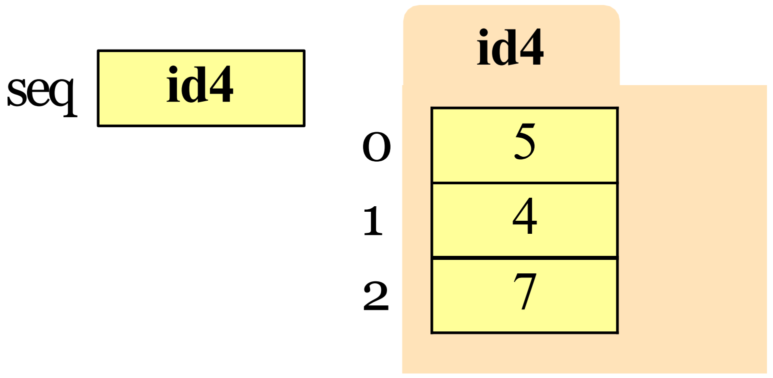


For Loops and Call Frames

```
1. def add_one(thelist):
2.     """Adds 1 to every elt
3.     Pre: thelist all nums"""
4.     for x in thelist:
5.         x = x+1
```

add_one(seq):





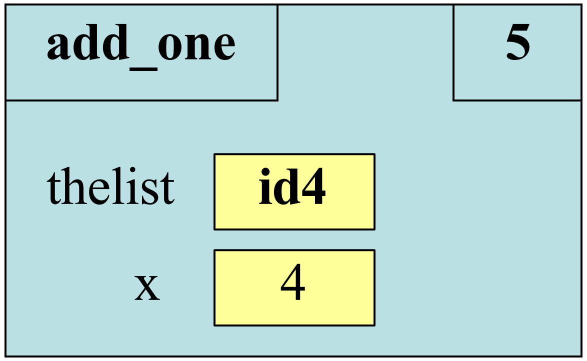
Increments x in **frame**

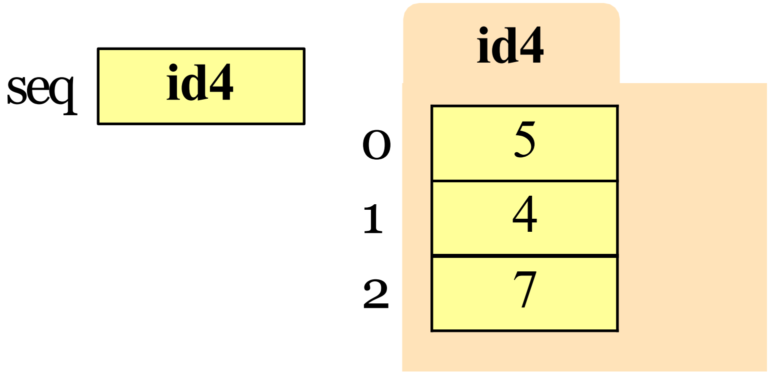
Does not affect folder

For Loops and Call Frames

```
1. def add_one(thelist):
2.     """Adds 1 to every elt
3.     Pre: thelist all nums"""
4.     for x in thelist:
5.         x = x+1
```

add_one(seq):



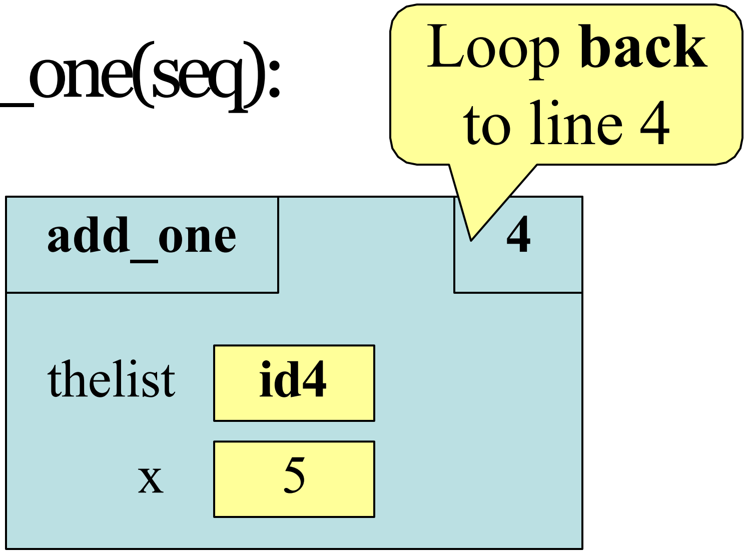


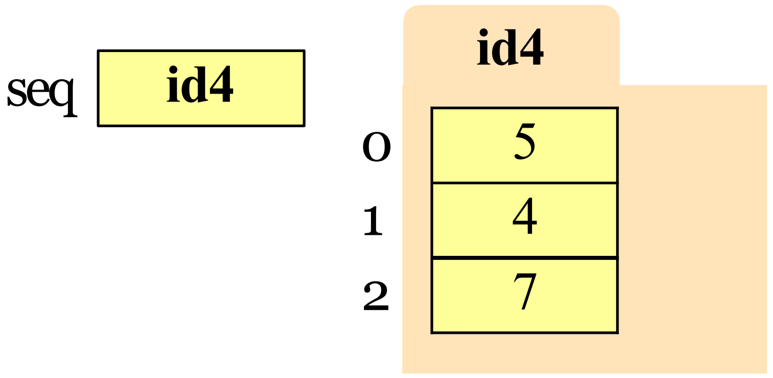
Next element stored in x. Previous calculation lost.

For Loops and Call Frames

```
1.  def add_one(thelist):
2.      """Adds 1 to every elt
3.      Pre: thelist all nums"""
4.      for x in thelist:
5.          x = x+1
```

add_one(seq):

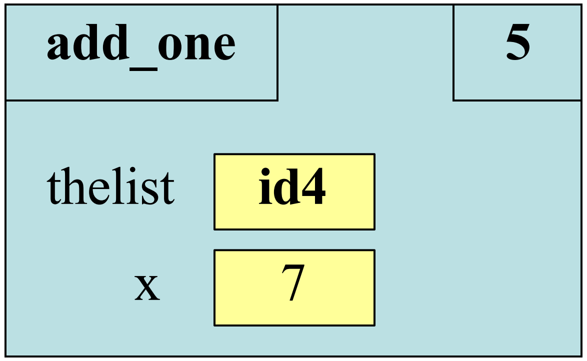


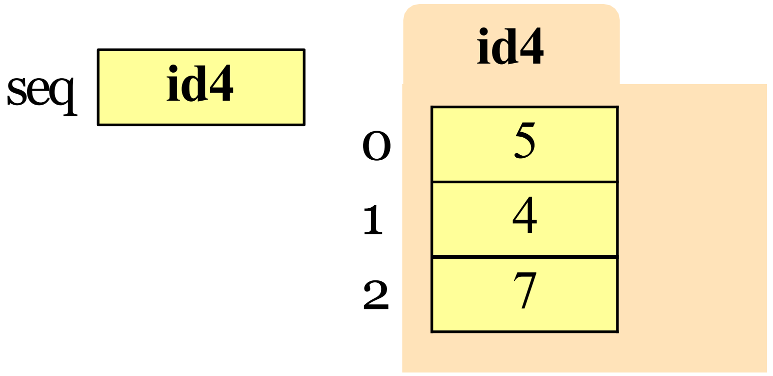


For Loops and Call Frames

1. def add_one(thelist):
2. """Adds 1 to every elt
3. **Pre**: thelist all nums"""
4. for x in thelist:
5. x = x+1

add_one(seq):



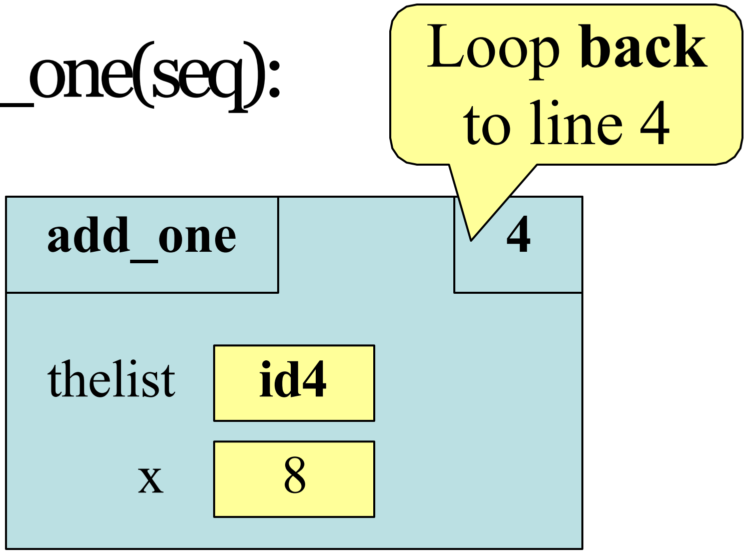


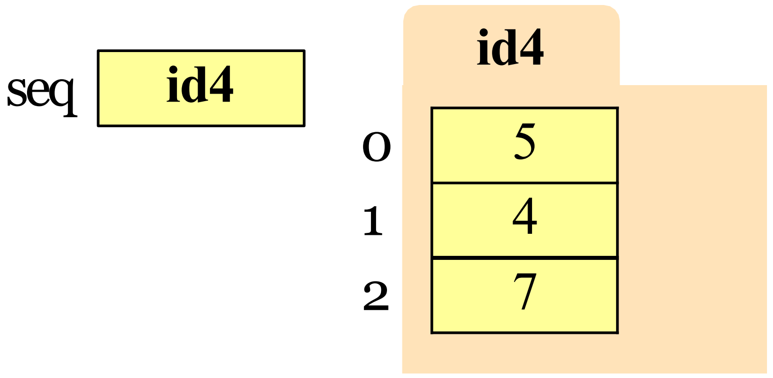
Next element stored in x.
Previous calculation lost.

For Loops and Call Frames

```
1.  def add_one(thelist):
2.      """Adds 1 to every elt
3.      Pre: thelist all nums"""
4.      for x in thelist:
5.          x = x+1
```

add_one(seq):

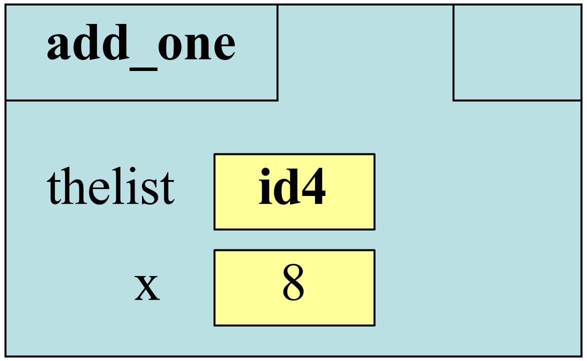


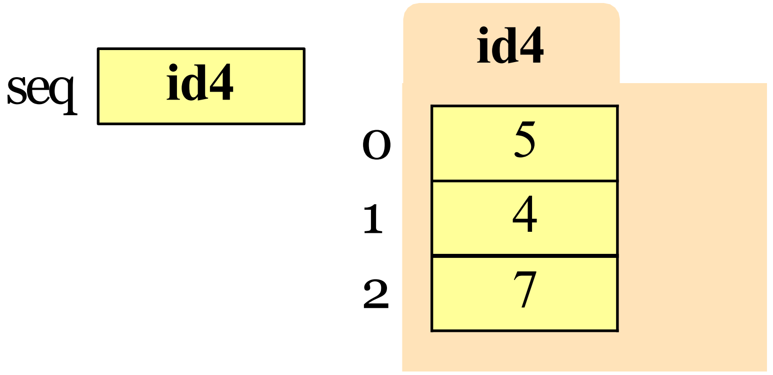


For Loops and Call Frames

```
1. def add_one(thelist):
2.     """Adds 1 to every elt
3.     Pre: thelist all nums"""
4.     for x in thelist:
5.         x = x+1
```

add_one(seq):





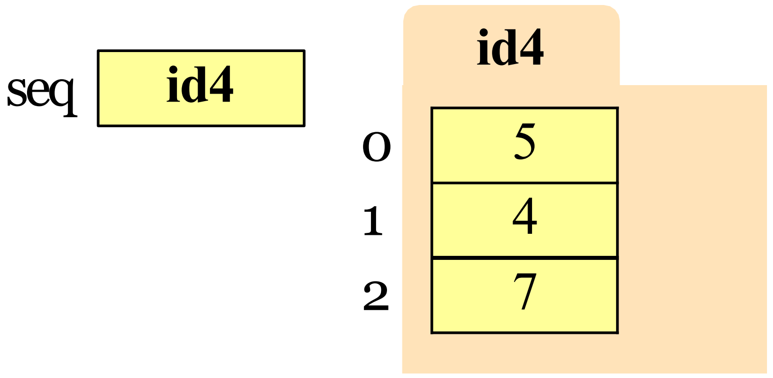
Loop is **completed.**
Nothing new put in x.

For Loops and Call Frames

```
1.  def add_one(thelist):
2.      """Adds 1 to every elt
3.      Pre: thelist all nums"""
4.      for x in thelist:
5.          x = x+1
```

add_one(seq):

ERASE WHOLE FRAME



No changes to folder

On The Other Hand

```
def copy_add_one(thelist):
    """Returns: copy with 1 added to every element
    Precondition: thelist is a list of all numbers
    (either floats or ints)"""
    mycopy = []  # accumulator
    for x in thelist:
        x = x+1
        mycopy.append(x)   # add to end of accumulator
    return mycopy
```

Accumulator keeps result from being lost

How Can We Modify A List?

- **Never** modify iterable!
- This is an infinite loop:

```
for x in thelist:
    thelist.append(1)
```

Try in Python Tutor to see what happens

- Need a second sequence
- How about the *positions*?

```
thelist = [5, 2, 7, 1]
thepos = [0, 1, 2, 3]

for x in thepos:
    thelist[x] = thelist[x]+1
```

How Can We Modify A List?

- **Never** modify iterable!
- This is an infinite loop:

```
for x in thelist:
    thelist.append(1)
```

Try in Python Tutor
to see what happens

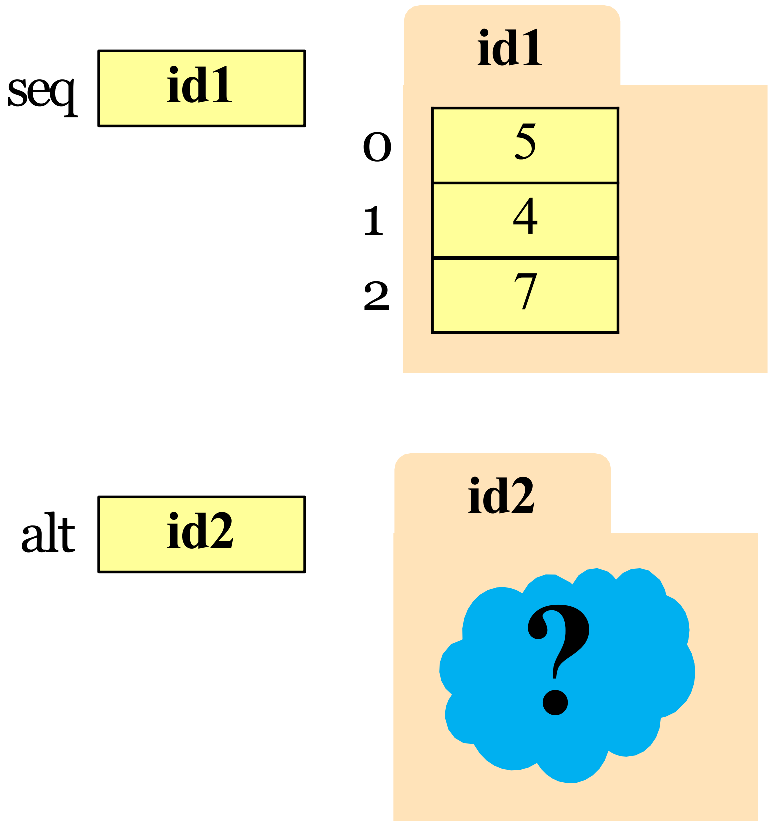
- Need a second sequence
- How about the *positions*?

```
thelist = [5, 2, 7, 1]
           ↑  ↑  ↑  ↑
thepos  = [0, 1, 2, 3]
```

```
for x in thepos:
    thelist[x] = thelist[x]+1
```

This is the Motivation for Iterables

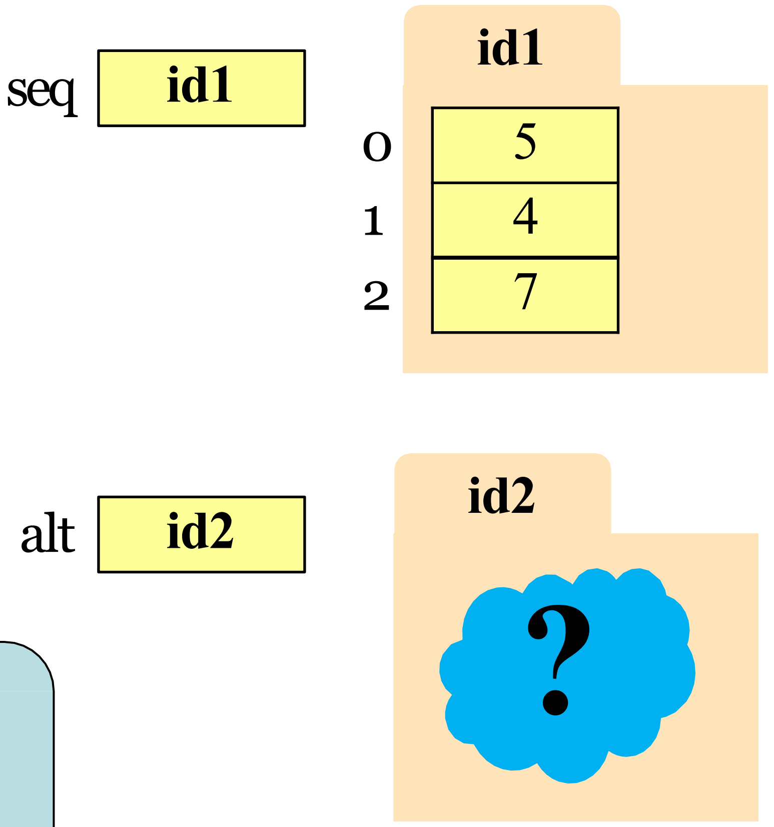
- **Iterables** are objects
  - Contain data like a list
  - **But cannot slice them**
- Have list-like properties
  - Can use then in a for-loop
  - Can convert them to lists
  - mylist = list(myiterable)
- **Example**: Files
  - Use open() to create object
  - Makes iterable for reading



Iterables, Lists, and For-Loops

```
>>> file = open('sample.txt')
>>> list(file)
['This is line 1\n',
'This is line 2\n']
>>> file = open('sample.txt')
>>> for line in file:
...       print(line)
This is line one

This is line two
```

print adds \n in *addition* to one from file



The Range Iterable

- range(x)
  - Creates an iterable
  - Stores [0,1,…,x-1]
  - **But not a list!**
  - But try list(range(x))
- range(a,b)
  - Stores [a,…,b-1]
- range(a,b,n)
  - Stores [a,a+n,…,b-1]

- Very versatile tool
- Great for processing ints

Accumulator

```
total = 0

# add the squares of ints
# in range 2..200 to total

for x in range(2,201):
    total = total + x*x
```

Modifying the Contents of a List

```
def add_one(thelist):
    """(Procedure) Adds 1 to every element in the list
    Precondition: thelist is a list of all numbers
    (either floats or ints)"""
    size = len(thelist)
    for k in range(size):
        thelist[k] = thelist[k]+1
    # procedure; no return
```

Iterator of list **positions** (safe)

WORKS!